Programación Funcional Avanzada

Haskell Concurrente

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia vs. Paralelismo

- **Concurrencia**
  - Múltiples tareas.
  - Simultáneas.
  - Posiblemente independientes.
  - Técnica para estructurar procesamiento.

Concurrencia vs. Paralelismo

- **Concurrencia**
  - Múltiples tareas.
  - Simultáneas.
  - Posiblemente independientes.
  - Técnica para estructurar procesamiento.
- **Paralelismo**
  - Una sola tarea.
  - Divisible en tareas parciales.
  - Resultado es la combinación.

Concurrencia vs. Paralelismo

- **Concurrencia**
  - Múltiples tareas.
  - Simultáneas.
  - Posiblemente independientes.
  - Técnica para estructurar procesamiento.
- **Paralelismo**
  - Una sola tarea.
  - Divisible en tareas parciales.
  - Resultado es la combinación.
- Ambos permiten mejorar el desempeño
  - Aprovechando múltiples *cores*.
  - Disimulando latencia en programas IO *bound*.

Haskell permite atacar ambos tipos de problemas
Hoy nos enfocamos en **concurrencia**

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia en Haskell

Paralelismo explícito con el modelo de hilos

- Concurrencia "ligera" – *Unbounded threads*
  - Creación de hilos y cambio de contexto muy baratos.
  - Controlada por el *Run Time System*.
  - Independiente del sistema operativo.

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia en Haskell

Paralelismo explícito con el modelo de hilos

- Concurrencia "ligera" – *Unbounded threads*
  - Creación de hilos y cambio de contexto muy baratos.
  - Controlada por el *Run Time System*.
  - Independiente del sistema operativo.
- Concurrencia "pesada" – *bounded threads*
  - Controlada por el sistema operativo.
  - Tan eficiente como los hilos en el sistema operativo anfitrión.

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia en Haskell

Paralelismo explícito con el modelo de hilos

- Concurrencia "ligera" – *Unbounded threads*
  - Creación de hilos y cambio de contexto muy baratos.
  - Controlada por el *Run Time System*.
  - Independiente del sistema operativo.
- Concurrencia "pesada" – *bounded threads*
  - Controlada por el sistema operativo.
  - Tan eficiente como los hilos en el sistema operativo anfitrión.
- Ambos aprovechan **todos** los *cores* disponibles.

Hilos "ligeros" son uno o dos órdenes de magnitud
más eficientes que los "pesados".

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia en Haskell

`Control.Concurrent`

- Creación de hilos "ligeros" o "pesados".

```
forkIO :: IO () -> IO ThreadId
forkOS :: IO () -> IO ThreadId
```

- `ThreadId` es un tipo abstracto opaco que modela hilos – convenientemente instanciado en `Eq`, `Ord` y `Show`
- La función `main` ejecuta en el hilo inicial.
- Hilos son clausuras – copian estado inmutable.
- *Scheduling* no determinístico – pero justo.
- *Pre-emptive multitasking* – ceden al reservar memoria.

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia en Haskell

`Control.Concurrent`

- Operaciones habituales para concurrencia

```
myThreadId :: IO ThreadId
threadDelay :: Int -> IO () -- miliseguntos
killThread :: ThreadId -> IO ()
yield :: IO ()
```

- Aprovechar múltiples núcleos

```
getNumCapabilities :: IO Int
forkOn :: Int -> IO () -> IO ThreadId
threadCapability :: ThreadId -> IO (Int,Bool)
```

- Lanzar excepciones selectivamente

```
throwTo :: (Exception e) => ThreadId -> e -> IO ()
```

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia en Haskell

```
import System.Directory
import Control.Concurrent

main = do
  forkIO $ writeFile "foo.txt" "Kilroy was here!"
  r <- doesFileExist "foo.txt"
  print r
```

- A veces `True`, a veces `False`.
- Solamente con un *core* – imposible predecir cuál hilo se queda con él.
- Más de un *core* – imposible predecir en que orden corren los hilos.

UNIVERSIDAD SIMÓN BOLÍVAR

Can I has *Sockets*?

Network

- Tipos de datos abstractos

```
data Socket      -- Eq, Show
data PortID      -- Service, PortNumber, UnixSocket
```

- Actuar como servidor

```
listenOn :: PortID -> IO Socket
accept   :: Socket ->
            IO (Handle, HostName, PortNumber)
sClose   :: Socket -> IO ()
```

- Actual como cliente

```
connectTo :: HostName -> PortID -> IO Socket
```

Se leen y escriben como archivos.
Buffered por omisión

UNIVERSIDAD SIMÓN BOLÍVAR

Un servidor `echo`

- Repite de vuelta lo que llega por el puerto (*socket* TCP).
- Si se corta la conexión (señal `SIGPIPE`), termina sin quejarse.

Un servidor echo

- Repite de vuelta lo que llega por el puerto (*socket* TCP).
- Si se corta la conexión (señal SIGPIPE), termina sin quejarse.

```
main = do
  installHandler sigPIPE Ignore Nothing
  s <- listenOn (PortNumber 9900)
  acceptConnections s

acceptConnections sock = do
  conn@(h,host,port) <- accept sock
  forkIO $ catch (talk conn `finally` hClose h)
                 ((\e :: SomeException) -> print e)
  acceptConnections sock

talk conn@(h,_,_) =
  hGetLine h >>= hPutStrLn h >> hFlush h >> talk conn
```

Comunicación entre hilos

- Hilos comparten estado inmutable – sin riesgo de colisión.
- `MVar` – tipo abstracto para *variables de sincronización*

```
newEmptyMVar :: IO a
newMVar      :: a -> IO (MVar a)
putMVar      :: MVar a -> a -> IO ()
takeMVar     :: Mvar a -> IO a
```

  - Contenedor con un sólo elemento – vacía o llena.
  - `putMVar` se bloquea si está llena.
  - `takeMVar` se bloquea si está vacía.
  - ¡No son `Functor` a propósito!

UNIVERSIDAD SIMÓN BOLÍVAR

Comunicación entre hilos

- Hilos comparten estado inmutable – sin riesgo de colisión.
- `MVar` – tipo abstracto para *variables de sincronización*

```
newEmptyMVar :: IO a
newMVar      :: a -> IO (MVar a)
putMVar      :: MVar a -> a -> IO ()
takeMVar     :: Mvar a -> IO a
```

  - Contenedor con un sólo elemento – vacía o llena.
  - `putMVar` se bloquea si está llena.
  - `takeMVar` se bloquea si está vacía.
  - ¡No son `Functor` a propósito!
- *Fair Runtime* – garantiza *single wakeup* FIFO de los hilos bloqueados.
- ¡Son perezosos! – Cuidado con guardar *thunks*.

UNIVERSIDAD SIMÓN BOLÍVAR

Combinando para comunicar

Compartir los nombres de las cajas

```
communicate = do
  m <- newEmptyMVar
  forkIO $ do
    v <- takeMVar m
    putStrLn $ "recibi: " ++ show v
  putStrLn "enviando"
  putMVar m "despierta!"
```

- Se comparte el nombre `m` que es inmutable . . .
- . . . pero que hace referencia a un `MVar` mutable.

Aprovechando `MVar`

- Usos obvios de un `MVar`
  - Pasar valores (¡mensajes!) entre hilos.
  - Usarlos como *mutex*.

Aprovechando `MVar`

- Usos obvios de un `MVar`
  - Pasar valores (¡mensajes!) entre hilos.
  - Usarlos como *mutex*.
- Evitar bloqueos sin necesidad.

```
tryTakeMVar :: MVar a -> IO (Maybe a)
tryPutMvar  :: MVar a -> a -> IO Bool
```

UNIVERSIDAD SIMÓN BOLÍVAR

Aprovechando `MVar`

- Usos obvios de un `MVar`
  - Pasar valores (¡mensajes!) entre hilos.
  - Usarlos como *mutex*.
- Evitar bloqueos sin necesidad.

```
tryTakeMVar :: MVar a -> IO (Maybe a)
tryPutMvar  :: MVar a -> a -> IO Bool
```

- Simplificar operaciones atómicas – incluso ante excepciones.

```
modifyMVar :: MVar a -> (a -> IO (a,b)) -> IO b
```

Esta última es interesante. . .

UNIVERSIDAD SIMÓN BOLÍVAR

Un patrón frecuente...

```
modifyMVar :: MVar a -> (a -> IO (a,b)) -> IO b
modifyMVar m op =
  block $ do
    a <- takeMVar m
    (b,r) <- unblock (op a)
             `catch`
             (\e -> putMVar m a >> throw e)
    putMVar m b
    return r
```

- `block`/`unblock` – impedir/permitir recepción de excepciones.
- `catch` – intenta la primera acción, y en caso de recibir una excepción invoca a la función de rescate.

Vienen de `Control.Exception`.
`mask` reemplaza a `block`/`unblock` en GHC 7

UNIVERSIDAD SIMÓN BOLÍVAR

Un *portscanner* concurrente

Tom Moertel

- Determina cuáles puertos están activos en un rango particular.
- Un hilo para analizar cada puerto.

```
main = do
  args <- getArgs
  case args of
    [host,from,to] -> scanRange host
                                [read from .. read to]
    _              -> usage

usage = do
  hPutStrLn stderr "Usage: Portscan host from to"
  exitFailure
```

UNIVERSIDAD SIMÓN BOLÍVAR

Un *portscanner* concurrente

```
scanRange host ports =
  mapM (threadWithChannel .
          scanPort host .
            fromIntegral) ports >>= mapM_ hitCheck
  where
    hitCheck mv = takeMVar mv >>= maybe (return ())
                                        printHit
    printHit port = putStrLn =<< showService port

threadWithChannel action = do
  mvar <- newEmptyMVar
  forkIO $ action >>= putMVar mvar
  return mvar
```

- Por cada puerto de la lista, generar un hilo para revisarlo.
- Un `MVar` por hilo para indicar éxito o fracaso con un `Maybe`.

UNIVERSIDAD SIMÓN BOLÍVAR

Un *portscanner* concurrente

```
scanPort host port =
  withDefault Nothing (tryPort >> return (Just port))
  where
    tryPort = connectTo host (PortNumber port)
                      >>= hClose

showService port =
  withDefault (show port) $ do
    service <- getServiceByPort port "tcp"
    return (show port ++ " " ++ serviceName service)

withDefault defaultVal action =
    handle (\(e :: SomeException) -> return defaultVal)
```

- Es un *portscan* "evidente" con connectTo.
- Determinar el nombre del servicio para mostrarlo.
- Hablaremos de excepciones otro día.

UNIVERSIDAD SIMÓN BOLÍVAR

Aprovechando los *cores*

. . . los de verdad, *hyperthreading* no cuenta

- Los programas compilados con `ghc` sólo usan un *core*.
  - RTS **sin** hilos externos.

    ```
    $ ghc -o program ...
    ```
  - Hilos Haskell ejecutan sobre un sólo hilo del sistema operativo.

UNIVERSIDAD SIMÓN BOLÍVAR

Aprovechando los *cores*

. . . los de verdad, *hyperthreading* no cuenta

- Los programas compilados con `ghc` sólo usan un *core*.
  - RTS **sin** hilos externos.

    ```
    $ ghc -o program ...
    ```
  - Hilos Haskell ejecutan sobre un sólo hilo del sistema operativo.
- Enlazar con *Threaded Run Time System* – ¡usar múltiples *cores*!
  - RTS **con** hilos externos.

    ```
    $ ghc -threaded -o program ...
    ```
  - Hilos Haskell ejecutan sobre varios hilos del sistema operativo.

UNIVERSIDAD SIMÓN BOLÍVAR

Aprovechando los *cores*

. . . los de verdad, *hyperthreading* no cuenta

- Los programas compilados con `ghc` sólo usan un *core*.
  - RTS **sin** hilos externos.

    ```
    $ ghc -o program ...
    ```
  - Hilos Haskell ejecutan sobre un sólo hilo del sistema operativo.
- Enlazar con *Threaded Run Time System* – ¡usar múltiples *cores*!
  - RTS **con** hilos externos.

    ```
    $ ghc -threaded -o program ...
    ```
  - Hilos Haskell ejecutan sobre varios hilos del sistema operativo.
- Ejecutar el programa indicando cuántos *cores* aprovechar

  ```
  $ program +RTS -N2 -RTS ...
  ```

`-N` usa todos los que encuentra

UNIVERSIDAD SIMÓN BOLÍVAR

¡Claro que funciona en GHCi!

- Usualmente no uso múltiples *cores* cuando desarrollo...

```
$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/
[...]
Prelude> GHC.Conc.numCapabilities
1
```

UNIVERSIDAD SIMÓN BOLÍVAR

¡Claro que funciona en GHCi!

- Usualmente no uso múltiples *cores* cuando desarrollo...

```
$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/
[...]
Prelude> GHC.Conc.numCapabilities
1
```

- ...¡pero cuando lo hago, los uso todos!

```
$ ghci +RTS -N -RTS
GHCi, version 7.6.3: http://www.haskell.org/ghc/
[...]
Prelude> GHC.Conc.numCapabilities
4
```

UNIVERSIDAD SIMÓN BOLÍVAR

De vuelta al *portscanner* concurrente

- Compilamos para aprovechar hilos

```
$ ghc -o portscan --make -threaded portscan.hs
```

De vuelta al *portscanner* concurrente

- Compilamos para aprovechar hilos

```
$ ghc -o portscan --make -threaded portscan.hs
```

- Lo corremos con un sólo *core* . . .

```
$ portscan localhost 1 1000
```

UNIVERSIDAD SIMÓN BOLÍVAR

De vuelta al *portscanner* concurrente

- Compilamos para aprovechar hilos

```
$ ghc -o portscan --make -threaded portscan.hs
```

- Lo corremos con un sólo *core* . . .

```
$ portscan localhost 1 1000
```

- . . . o varios *cores*

```
$ portscan +RTS -N2 -RTS localhost 1 1000
```

UNIVERSIDAD SIMÓN BOLÍVAR

De vuelta al *portscanner* concurrente

- Compilamos para aprovechar hilos

```
$ ghc -o portscan --make -threaded portscan.hs
```

- Lo corremos con un sólo *core* . . .

```
$ portscan localhost 1 1000
```

- . . . o varios *cores*

```
$ portscan +RTS -N2 -RTS localhost 1 1000
```

- El resultado es igual – cambia la velocidad

```
$ portscan localhost 1 1000
22 ssh
80 www
631 ipp
```

UNIVERSIDAD SIMÓN BOLÍVAR

Canales

- Método alterno de comunicación entre hilos
  - Unidireccional.
  - Secuencia homogénea de valores – *stream*.
  - Tamaño limitado por la memoria disponible.

Canales

- Método alterno de comunicación entre hilos
  - Unidireccional.
  - Secuencia homogénea de valores – *stream*.
  - Tamaño limitado por la memoria disponible.
- Tipo abstracto para *canales*

```
newChan   :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan  :: Chan a -> IO a
```

  - `writeChan` **siempre** tiene éxito.
  - `readChan` se bloquea si no hay valores.

UNIVERSIDAD SIMÓN BOLÍVAR

Canales

- Método alterno de comunicación entre hilos
  - Unidireccional.
  - Secuencia homogénea de valores – *stream*.
  - Tamaño limitado por la memoria disponible.
- Tipo abstracto para *canales*

```
newChan   :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan  :: Chan a -> IO a
```

  - `writeChan` **siempre** tiene éxito.
  - `readChan` se bloquea si no hay valores.
- Pueden leerse como una lista perezosa (*lazy future*)

```
getChanContents :: Chan a -> IO [a]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Un ejemplo simple

```
import Control.Concurrent
import Control.Monad

main = do
  ch <- newChan
  forkIO (worker ch)
  xs <- getChanContents ch
  mapM_ putStr xs

worker ch = forever $ do
  v <- readFile "/proc/loadavg"
  writeChan ch v
  threadDelay (10^6)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Productor – Consumidor

```
import Control.Concurrent
import Control.Monad
import Data.Char

main = do
  ch <- newChan
  cs <- getChanContents ch
  forkIO $ producer ch
  consumer cs

producer c = forever $ do
  key <- getChar
  writeChan c key

consumer = mapM_ putChar . map shift
  where shift c | isAlpha c = chr (ord c + 1)
                | otherwise = c
```

A tener en cuenta ...

- Estructuras similares a las disponibles en lenguajes imperativos – mucho más seguras en términos de tipos y estructura del código.

A tener en cuenta ...

- Estructuras similares a las disponibles en lenguajes imperativos – mucho más seguras en términos de tipos y estructura del código.
- `MVar` y `Chan` son perezosos
  - Ambos procesan *thunks*
  - "Meter" algo en un `MVar` no necesariamente lo evalúa.

UNIVERSIDAD SIMÓN BOLÍVAR

A tener en cuenta . . .

- Estructuras similares a las disponibles en lenguajes imperativos – mucho más seguras en términos de tipos y estructura del código.
- `MVar` y `Chan` son perezosos
  - Ambos procesan *thunks*
  - "Meter" algo en un `MVar` no necesariamente lo evalúa.
- Sigue siendo difícil escribir programas correctos
  - *Deadlock* – por inversión de orden.
  - *Starvation* – por evaluación diferida.

Resolveremos esos problemas en la próxima clase.

UNIVERSIDAD SIMÓN BOLÍVAR

Let's go meta...

Look ma, no `IO` tricks!

- Vamos a modelar unos hilos muy ligeros...

```
data Thread = Print Char Thread
            | Fork Thread Thread
            | End
            deriving (Show,Eq)
```

Let's go meta...

Look ma, no `IO` tricks!

- Vamos a modelar unos hilos muy ligeros...

```
data Thread = Print Char Thread
            | Fork Thread Thread
            | End
            deriving (Show,Eq)
```

- ...y un tipo que contiene una función "curiosa"

```
newtype WTF a =
    WTF { fromWTF :: ((a -> Thread) -> Thread) }
```

  - La función recibe una función que produce un hilo...
  - ...y lo uso para producir otro hilo.
  - a es el "prefijo" de un hilo que existe –
    "lo que ha *corrido* hasta ahora" o "el presente"
  - Esas funciones construyen *futuros*.

UNIVERSIDAD SIMÓN BOLÍVAR

Let's keep going meta...

¿Cómo puedo construir *combinaciones* de futuros?

Let's keep going meta...

¿Cómo puedo construir *combinaciones* de futuros?

```
instance Monad WTF where
  return x = WTF $ \k -> k x
  m >>= f  = WTF $
             \k -> fromWTF m (\x -> fromWTF (f x) k)
```

Let's keep going meta...

¿Cómo puedo construir *combinaciones* de futuros?

```
instance Monad WTF where
  return x = WTF $ \k -> k x
  m >>= f  = WTF $
             \k -> fromWTF m (\x -> fromWTF (f x) k)
```

- Inyectar un valor en el futuro `k` es simplemente eso.

UNIVERSIDAD SIMÓN BOLÍVAR

Let's keep going meta...

¿Cómo puedo construir *combinaciones* de futuros?

```
instance Monad WTF where
  return x = WTF $ \k -> k x
  m >>= f  = WTF $
             \k -> fromWTF m (\x -> fromWTF (f x) k)
```

- Inyectar un valor en el futuro `k` es simplemente eso.
- ¿Cómo los combino?
  - `m` contiene un futuro.
  - `f` genera otro futuro a partir del presente `x`.
  - Entonces necesito construir un nuevo futuro, que comience con el futuro `m` y continúe con el futuro producido por `f x`

UNIVERSIDAD SIMÓN BOLÍVAR

Let's keep going meta...

¿Cómo puedo construir *combinaciones* de futuros?

```
instance Monad WTF where
  return x = WTF $ \k -> k x
  m >>= f  = WTF $
             \k -> fromWTF m (\x -> fromWTF (f x) k)
```

- Inyectar un valor en el futuro `k` es simplemente eso.
- ¿Cómo los combino?
  - `m` contiene un futuro.
  - `f` genera otro futuro a partir del presente `x`.
  - Entonces necesito construir un nuevo futuro, que comience con el futuro `m` y continúe con el futuro producido por `f x`

Epic future refuturing is epic!
Wait... What?

UNIVERSIDAD SIMÓN BOLÍVAR

Constructores de futuros

- Seguro está el final (death and taxes)

```
thread :: WTF a -> Thread
thread m = fromWTF m (const End)
```

Constructores de futuros

- Seguro está el final (death and taxes)

```
thread :: WTF a -> Thread
thread m = fromWTF m (const End)
```

- El futuro después de imprimir, es el mismo que antes.

```
cPrint :: Char -> WTF ()
cPrint c = WTF $ \k -> Print c (k ())
```

UNIVERSIDAD SIMÓN BOLÍVAR

Constructores de futuros

- Seguro está el final (death and taxes)

```
thread :: WTF a -> Thread
thread m = fromWTF m (const End)
```

- El futuro después de imprimir, es el mismo que antes.

```
cPrint :: Char -> WTF ()
cPrint c = WTF $ \k -> Print c (k ())
```

- It's dead, Jim!

```
cEnd :: WTF a
cEnd = WTF $ \_ -> End
```

UNIVERSIDAD SIMÓN BOLÍVAR

Constructores de futuros

- Seguro está el final (death and taxes)

```
thread :: WTF a -> Thread
thread m = fromWTF m (const End)
```

- El futuro después de imprimir, es el mismo que antes.

```
cPrint :: Char -> WTF ()
cPrint c = WTF $ \k -> Print c (k ())
```

- It's dead, Jim!

```
cEnd :: WTF a
cEnd = WTF $ \_ -> End
```

- El futuro de un *fork* es su problema, nosotros seguimos el nuestro.

```
cFork :: WTF a -> WTF ()
cFork m = WTF $ \k -> Fork (thread m) (k ())
```

UNIVERSIDAD SIMÓN BOLÍVAR

Scheduler de pobre

```
type Output      = [Char]
type ThreadQueue = [Thread]

runCM :: WTF a -> Output
runCM m = dispatch [] (thread m)

dispatch :: ThreadQueue -> Thread -> Output
dispatch rq (Print c t)  = c : schedule (rq ++ [t])
dispatch rq (Fork t1 t2) = schedule (rq ++ [t1,t2])
dispatch rq End          = schedule rq

schedule :: ThreadQueue -> Output
schedule []     = []
schedule (t:ts) = dispatch ts t
```

UNIVERSIDAD SIMÓN BOLÍVAR

Dos hilos tontos, pero efectistas

```
p1 :: WTF ()
p1 = do cPrint '1'
        cPrint '2'
        cPrint '3'
        cPrint '4'
        cPrint '5'
        cPrint '6'
        cPrint '7'
        cPrint '8'
        cPrint '9'
```

```
p2 :: WTF ()
p2 = do cPrint 'a'
        cPrint 'b'
        cPrint 'c'
        cPrint 'd'
        cPrint 'e'
        cPrint 'f'
        cPrint 'g'
        cPrint 'h'
        cPrint 'i'
```

UNIVERSIDAD SIMÓN BOLÍVAR

Y el programa principal

```
p3 :: WTF ()
p3 = cFork p1 >> cPrint '!' >> cFork p2 >> cPrint '?'

main = print $ runCM p3
```

UNIVERSIDAD SIMÓN BOLÍVAR

Quiero saber más...

- Documentación de `Control.Concurrent`
- Documentación de `Network`
- Documentación de `Control.Exception`

UNIVERSIDAD SIMÓN BOLÍVAR